Photo Sharing Pub/Sub (PSPS) Project Report

Group 5: Taylor Futral, Karen Chiao, Alex Cheng

Introduction

Many companies, old and new, are built around image storage and distribution. Instagram alone is worth over $100 billion. Images are typically larger in data-size than most messages, so the scale of transmission and storage presents a unique challenge. These challenges are compounded when making an application with millions or billions of world-wide users. We look to explore these challenges by creating our own Distributed Image storage system. Our goal is to create an application somewhat similar to Instagram, where users can share and view photos based on various topics.

To accomplish this we build a web platform oriented around the idea of users being able to pull in an image based on a topic. Once there are photos are uploaded, users will be able to "subscribe" to various topics. Our system, through the use of Kafka, is able to deal with load balancing the distributed work of handling multiple users and multiple images.

Overview of Research

This project is based on the idea of a Publish/Subscribe. Before creating our system we had to break down different ways of implementation including different ways of load balancing. There are three main types of Publish/Subscribe systems that we will talk about in this paper. The first being Topic-based Publish Subscribe, this is talked about in "The Many Faces of Publish/Subscribe" [2] and "On the fly load balancing to address hot topics in topic-based"[5] . Then there is Content-based Publish Subscribe which is also talked about in the same paper [2]. Finally Type-based Publish Subscribe is talked about which is discussed in both "The Many Faces of Publish/Subscribe". After types of Publish Subscribe systems have been established, we can look further into how load balancing is handled within each type and in general. There are three main types of load balancing that are discussed, the first being optimal load balancing in broker networks [1]. The next is dynamic load balancing in clustering publish/subscribe systems [3]. Finally we talk about load balancing with broker networks but in the context of handling big data rather than finding a more optimal path.

These papers were perfect for discovering what could be done in terms of creating publish subscribe system for our project. There are many ways to implement a pub/sub system and each have their own advantages and disadvantages depending on what your system's needs are. Exploring these papers revealed what the pros and cons were, and how they would affect our systems setup and performance.

Topic Based Publish/Subscribe

Topic Based Publish/Subscribe was the first Pub/Sub system created. It was based around the idea of topics or subjects and has been implemented in many industry solutions. It took the notion of channels and used to bundle messaging between peers and extend it to characterize and classify content. Individuals can subscribe/publish content based around *keywords* which help organize the information. This is similar to the idea of groups when acting as a group communication in topic based roots. It is also often used for replication. Subscription to a topic can be seen as becoming a member of a group of that topic, and publishing an event on topic translates accordingly into broadcasting that event amongst the members of that topic.

While the concept of topic based publish/subscribe is easy to understand there are improvements that are offered that can add to its complexity. One of the more notable improvements of these is the use of hierarchies to coordinate topics. This allows the system to permit programmers to organize topics according to containment relationships and then some. This makes it so that a subscription to one topic cascades throughout the subtopics of a system along with it. These topics follow a URL like notation. An even more complex version of this is the some topics can be categorized by multiple other topics and keywords and combinations of such. This allows for more complex and intimate interactions between the users and the topics. This system seems fitting for something like and photo based pub/sub system.

Pub/Sub Types and Photo Sharing Pub/Sub

Given the many advantages and disadvantages of the differing types of publish/subscribe system. The question really comes down to two things, what is the most realistic publish/subscribe system for our Photo Sharing Pub/Sub (PSPS) project and what works best given the type of load balancing we are interested in implementing in our project. Without taking into account load balancing the best two choices to debate between seem to be Topic based pub/sub and Type based pub/sub. Topic based pub/sub has the pro of being very simple in nature. The fact that it operates around *keywords*. This perfectly fits the scope of our project because the users can just subscribe to whatever keyword they request and then we can provide the paths for whatever they choose. This of course comes with the disadvantage of limited expressiveness as explained earlier, however this may not be a necessary feature in our application. If limited expressiveness is an issue, Type based pub/sub is capable of being fully expressive without the limitations seen in Topic based pub/subs. Within Type based the notion of event kind is directly matched with that of event type. As explained above, this means that type-based naturally describes the content-based filtering through public members of the considered event type. This allows for a deeper and a more intimate user experience for our app users.

After understanding the main concepts of how content is generally managed in the three pub/sub systems, we take a look a different load balancing techniques that occur as a result of different system architectures. The most common types are Tree-based and DHT-based, which older papers did not include much discussion on the issue of load balancing. We study an example of a DHT-based system that has load balancing as well as a cluster-based system and the components that are load balanced in it.

Cluster-Based Load Balancing

The cluster-based pub/sub system introduced by [3] organizes the brokers into clusters, and each node in a cluster is connected to nodes in other clusters to form a ring-shaped structure. The brokers support subscription dissemination and event dissemination. When a new subscription arrives at a broker, it is added to a local list (removing other subscriptions that the new one covers) and is sent to all other brokers in the same cluster, who will check the cluster's subscription list and apply the same methodology as the first broker to their cluster list. When a new event arrives at a broker, the broker broadcasts the event to the the other members in its ring, and those ring members will then broadcast the event to the members in their respective clusters. The loads that are managed by the brokers are client load (which is essentially the load due to a connected client) and forwarding load. The three components of this latter type of load are what is mainly balanced.

The forwarding load, which deals with passing publications and subscriptions between brokers), has three types: ring publish load, cluster publish load, and cluster subscription load.

The system dynamically load balances the three kinds of load with different algorithms. The overarching theme across the three is: when a node is overloaded, send the overloaded work to another node in the same cluster but has a underloaded ring. To prevent overloading other nodes, the overloaded brokers with cluster publish loads first ask for the help and send the overload once some node gives the okay, and the overloaded brokers with an incoming cluster subscription load will ask the disseminator to pause, while that node (and the node in its cluster that it shares the load with) do constant matching. When the broker is no longer overloaded after the pause, it sends a message to the disseminator to let it know it can receive new subscriptions.

This structure with its dynamic load balancing property in its brokers outperformed previous Tree and DHT based architectures. Due to the ring structure, it is resilient to broker failures and has fast content dissemination because of the reduced number of hopes between subscribers and use of parallelization on content matching.

Topic-based Load Balancing

We investigated a paper titled “On the fly load balancing to address hot topics in topic-based pub/sub systems” (Dedousis et al.) [5] that presents a an algorithm for pub/sub load balancing on distributed brokers. Each broker is assigned a set of partitions of a topic, which is distributed to its subscribers. This system works in two steps: (1) Use an algorithm to detect an overloaded broker. (2) Use another algorithm to reassign brokers to less loaded brokers. If a large proportion of the brokers are overloaded, then the entire system may be overloaded, the issue is bigger than this paper, which deals with an imbalanced load.

The first step to the system is identifying overloaded brokers, the paper presents two techniques: (1) If a particular broker’s traffic exceeds the average by some factor of the standard deviation, then the broker is overloaded. This factor must be chosen, thus this technique can be difficult to tune. (2) If a particular broker’s Local Outlier Factor (LOF) is greater than 2, then we identify this broker as overloaded. There is a series of equations for calculating the LOF presented in this paper.

After identifying the overloaded brokers, we now need to determine which brokers should accept new partitions. In this paper, the algorithm to do so is presented here: the most overloaded broker selects the least overloaded broker, and steps through a sorted list of all the brokers with a step size equal to the number of overloaded brokers until reaching the position of the overloaded brokers. The second most overloaded broker selects the second least overloaded broker and steps through like the first one, and so on. This process generates a migration broker set (MB), which also includes the original overloaded broker. For example, if brokers 1,2,3 are overloaded out of 4,5,6,7,8,9, then 1’s migration broker set is {9, 6}, 2 maps to {8, 5}, lastly 3 maps to {7, 4}.

After creating this migration broker set, we now need need to determine which partitions should be moved to these new partitions. Once the migration broker set has been determined, the authors propose an optimization algorithm that seeks to minimize the maximum load on the set of brokers accepting new partitions. The output of this algorithm is a matrix of boolean values that map old partitions in the overloaded broker to new brokers. This algorithm is an NP-complete problem, which ultimate is equivalent to to the “multiprocessor scheduling problem” This problem can be solved using the Longest Processing Time” algorithm, which has an upper bound of 4/3-1/(3B), where B is the number of brokers in the migration broker set. When performing this load-balancing, we must also check that no additional brokers become overloaded. If other brokers become overloaded, then we will most likely need to increase the

number of brokers. The paper successfully implemented this load balancing system to achieve a balanced situation where no node was overloaded after running the Kafka system.

Overall System Architecture

Taking into account the different types of publish/subscribe systems and the load balancing techniques available, we find that Topic based load balancing along with Clustering worked best for Photo Sharing Pub/Sub. This is because with Topic based load balancing individuals can subscribe/publish based around topics that they want to define or have already been defined. This is simple in nature and worked well with what we wanted to implement. This also allowed us to focus on the more important aspect of our system which is load balancing. For load balancing we found that a Clustering based system worked best because the ring structure described in Dynamic Load Balancing for Cluster-based Publish/Subscribe System [3] works well at redistributing work and is tolerant to network failures. One of the concerns described by Jafarpour [3] is the load balancing of users, and faulty connections could possibly be dealt with.

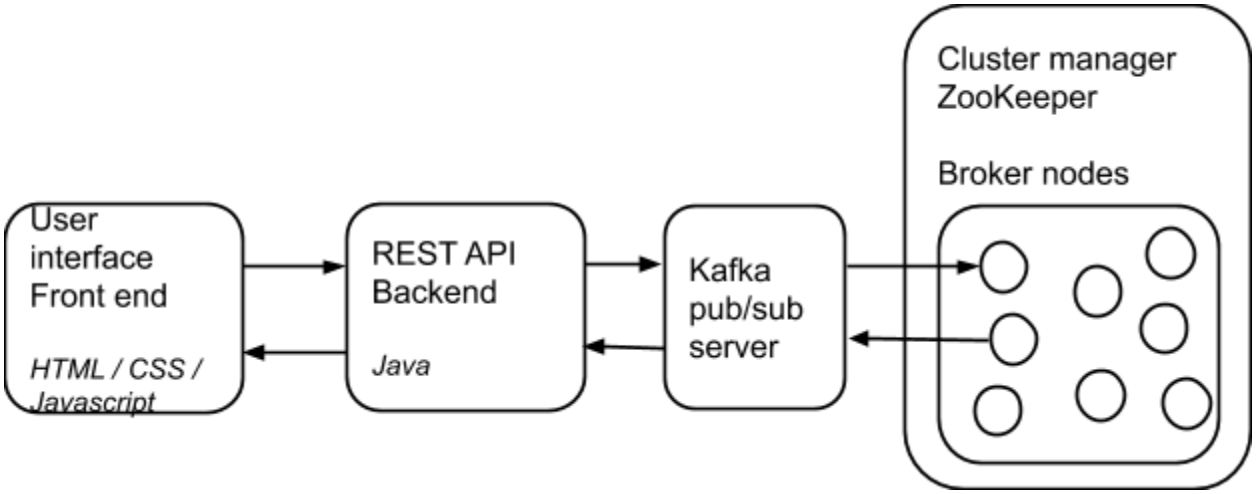


Image of system architecture

The backbone of our cluster-based system is Apache Kafka [7] and Apache Zookeeper and we have developed a front-end interface for users to access our system and send images as their messages rather than strings of text. Apache Kafka is a publish and subscribe messaging system that has existing APIs for developers to plug-and-play. In the latest versions of Kafka, Zookeeper is a key integration that acts as the managing service for the cluster of broker nodes which handle and pass the messages. Users on the front end are able to publish images to new or existing topics and also subscribe to image feeds of available topics that pique their interest.

Implementation

A large majority of the time to create this system was spent understanding Kafka and how it works. We found that Kafka has a steep learning curve. The importance of using Kafka was its support of providing an architecture for splitting topics into partitions. Having multiple partitions, or in-sync copies, of data for a topic allows for more users to actively read from the data. Each user maintains a consumer_offset value for consuming data in parallel. The importance of Zookeeper is its distributed architecture and pre-built algorithms that facilitate the coordination of the broker nodes and user traffic.

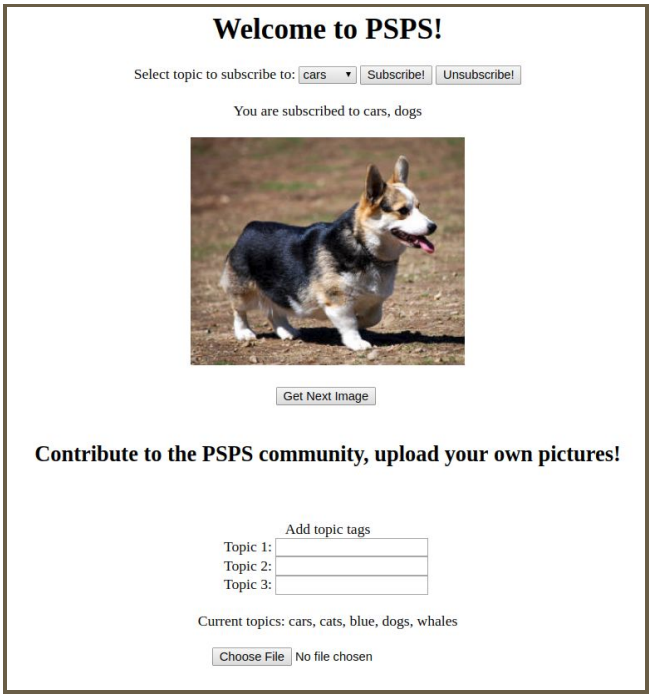


Image of PSPS end user experience

Frontend

Our frontend allows users to view and upload pictures to the Photo Sharing Pub Sub application. It uses the standard HTML / CSS / Javascript as well as JQuery for document manipulation and Ajax for API calls to our backend. To browse pictures, pick topics to subscribe to, then press a single button to cycle through pictures in those topics. Users can also upload pictures and specify up to 3 topics that picture belongs to. Topics are created when users specify a new topic. A demonstration of the front-end can be found here: https://www.youtube.com/watch?v=fVXw8kCJ0iU

Backend

The backend is a REST API built using Java. We used Grizzly to host our server, and Jersey to map URLs to endpoints. The backend acts as an interface between the user and the Kafka server. For example, when a user uploads a picture, the backend uses the Kafka Producer API to send data to the Kafka server. Similarly for subscribing, the backend uses the Kafka Consumer API to fetch data from the Kafka server, before packaging it and serving it to the frontend as JSON or JPG.

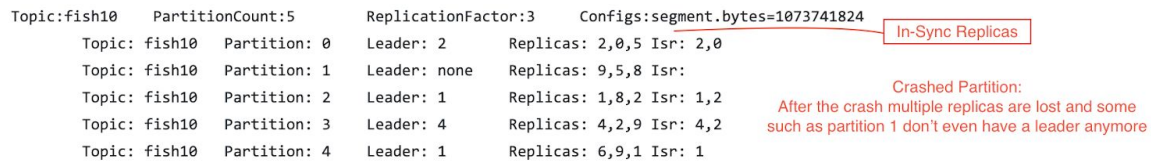
Results

During a normal multi-user use case, the system was able to support concurrents reads and concurrent writes (see Demo) without any noticeable delays or conflicts.

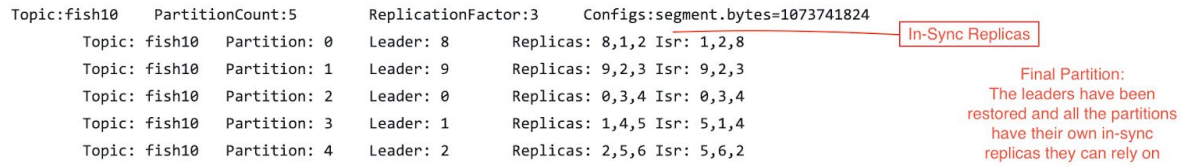
The main test we ran dealt with load balancing topic partitions. We ran a network of 10 brokers that supported 10 topics, where each topic set to have 5 partitions and 3 replicas. Upon initialization, the topic partitions with their replicas were evenly distributed across all of the brokers. To simulate a failure in the network, we manually shut down 4 nodes, thus leaving only 60% of the brokers still functional. We found that the internal Zookeeper algorithm supports

quick reassignment of the leaders for the partitions, so almost all 10 topics and their 5 partitions found new brokers as their leaders. There would be instances where a partition would be unable to find a new leader since its leader node and its replicas happened to have all gone down with the failure. After verifying the reassignment, the crashed nodes were brought back online and reinstated to their original IDs, but we found that the topic partitions do not rebalance to the new empty nodes. This means that 60% of the network is supporting all of the traffic (e.g. allocating topic partitions, updating partitions and replicas with publisher uploads, supporting multiple consumer downloads). We found that Zookeeper provides a script that suggests possible reassignments of topic partitions that can be quickly applied and executed to evenly redistribute the traffic to the newly added broker nodes. It appears that zookeeper uses a RoundRobin approach for its rebalancing algorithms, but it is not clear due to Zookeeper's abstraction under Kafka's abstraction.

The images below show the result of a topic after the simulated crash and after the reassignment script is executed. We can see that all topic partitions have a leader, since without a leader, a partition would not get updates. All topics also have three in-sync replicas (Isr) which indicate a healthy balance and distribution of topic partitions and their copies for supporting many users.



Crashed Partition



After Reassigning Topic Partition

Conclusion/Future Work

Although the initial set up took longer than preferred, Kafka proved to be a very impressive tool for implementing a publish subscribe system. If given another quarter with the system, we would like to spend more time understanding the different ways to manipulate Kafka. We would also maybe take the time to create better automated tests for our backend and expand on the features in our front end, since a huge challenge was finding the correct kafka scripts to run. Overall we learned that Kafka is a powerful publish and subscribe system with a dynamic and robust load balancing system.

References:

[1] Samrat, Ganguly, et al. Optimal Load Balancing in Publish/Subscribe Broker Networks (2008) citeseerx.ist.psu.edu

[2] Eugster, Patrick Th, et al. The Many Faces of Publish/Subscribe. infoscience.epfl.ch/record/165428/files/10.1.1.10.1076.pdf.

[3] Jafarpour, Hojjat, et al. Dynamic Load Balancing for Cluster-based Publish/Subscribe System (2009) ics.uci.edu/~hjafarpo/Files/IEEE_SAINT09_CameraReady.pdf

[4] Nguyen, Hang Thi Thu et al. Load Balancing in Big Active Data (BAD) (2017) ics.uci.edu/~cs237/all_past_years/oldprojects/ProjectSlidesSpring2017/22Report.pdf

[5] Dedousis, Dimitris et al. On the fly load balancing to address hot topics in topic-based pub/sub systems (2018) https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=8416281

[6] João Paulo de Araujo et al. VCube-PS: A causal broadcast topic-based publish/subscribe system (2019)

[7] Jay Kreps et al. Kafka: a Distributed Messaging System for Log Processing (2011)

[8] Bunrong Leang et al. Improvement of Kafka Streaming Using Partition and Multi-Threading in Big Data Environment (2019)

[9] Ye Zhao, Nalini Venkatasubramanian et al. DYNATOPS: A Dynamic Topic-based Publish/Subscribe Architecture

[10] Yuuichi Teranishi et al. Scalable and Locality-Aware Distributed Topic-Based Pub/Sub Messaging for IoT (2016)